Koushik Sen, Darko Marinov, Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
{ksen,marinov,agha}@cs.uiuc.edu

ABSTRACT

In *unit testing*, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are *memory graphs*. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining *symbolic* and *concrete execution*, and more specifically, using such a combination to *generate test inputs* to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability,Verification

Keywords: concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs.

1. INTRODUCTION

Unit testing is a method for modular testing of a programs' functional behavior. A program is decomposed into units, where each unit is a collection of functions, and the units are independently tested. Such testing requires specification of values for the inputs (or *test inputs*) to the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or *test coverage*), several techniques have been proposed to automatically generate values for the inputs. One such technique is to *randomly* choose the values over the domain of potential inputs [4,8,10,21]. The problem with such random testing is two fold: first, many sets of values may lead to the same observable behavior and are thus *redundant*, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [20].

One approach which addresses the problem of redundant executions and increases test coverage is *symbolic execution* [1,3,9,22,23,27,28,30]. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an *execution path*. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking [14]. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To the best of our knowledge, Larson and Austin were the first to propose combining concrete and symbolic execution [16]. In their approach, the program is executed on some user-provided concrete input values. Symbolic path constraints are generated for the specific execution. These constraints are solved, if feasible, to see whether there are potential input values that would have led to a violation along the same execution path. This improves coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths.

Godefroid et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [11]. In Godefroid et al.'s approach, during a concrete execution, a conjunction of symbolic constraints along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate further test inputs which would direct the program along alternative paths. Specifically, they systematically negate the conjuncts in the path constraint to provide a depth first exploration of all paths in the computation tree. If it is not feasible to solve the modified constraints, Godefroid et al. propose simply substituting random concrete values.

A challenge in applying Godefroid et al.'s approach is to provide methods which extract and solve the constraints generated by a program. This problem is particularly com-

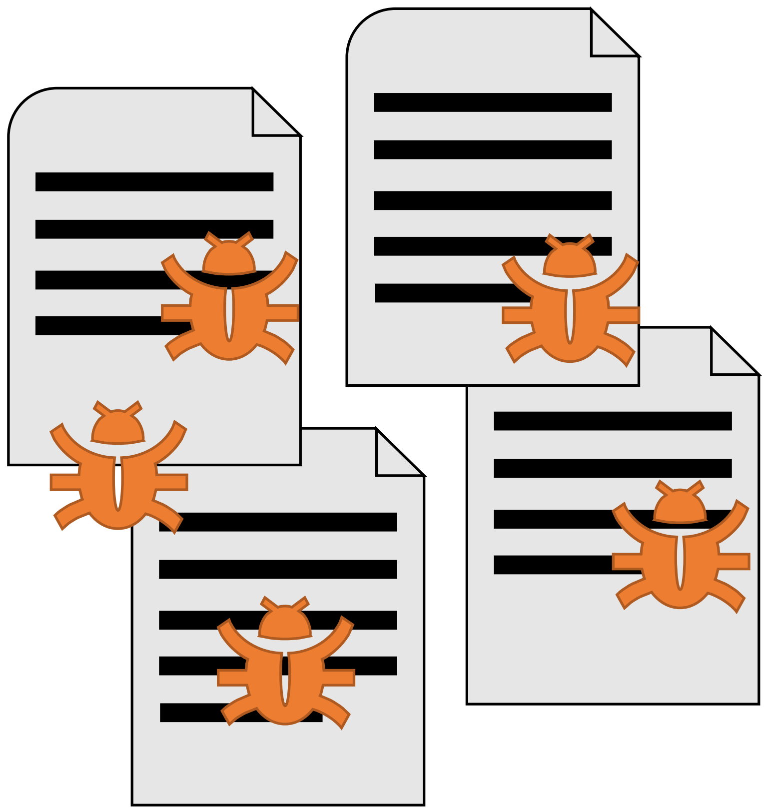
CUTE: A Concolic Unit Testing Engine for C (ACM SIGSOFT Impact Award 2019)

Koushik Sen, UC Berkeley

Darko Marinov, UIUC

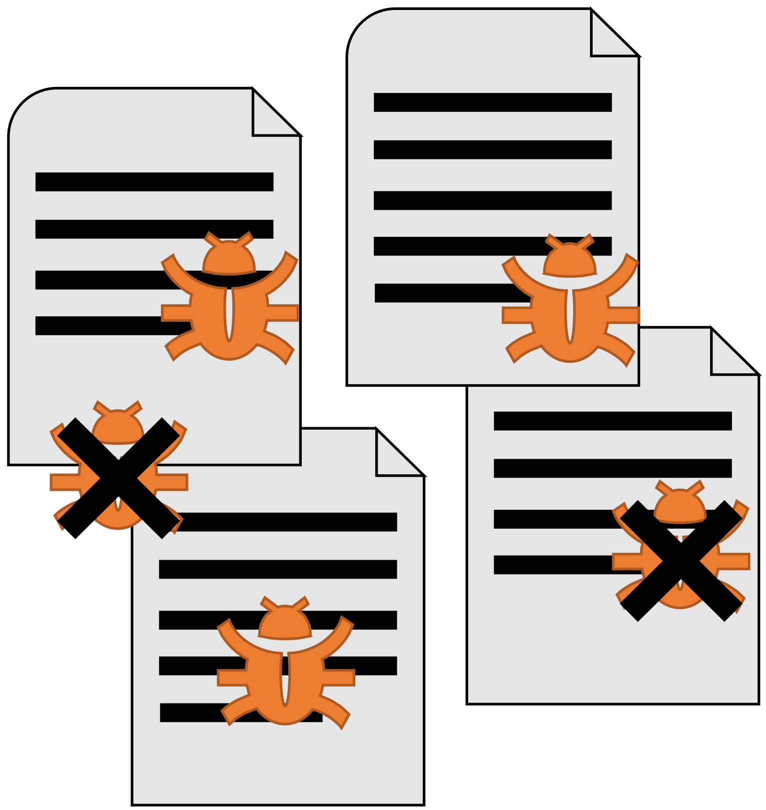
Gul Agha, UIUC

Programs Have Bugs

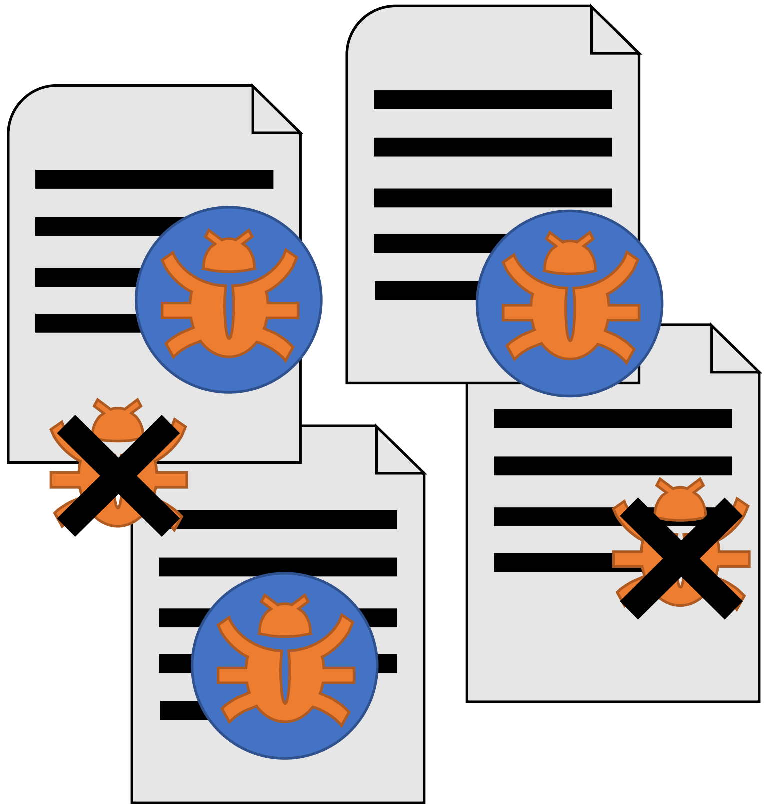


Why Program Testing?

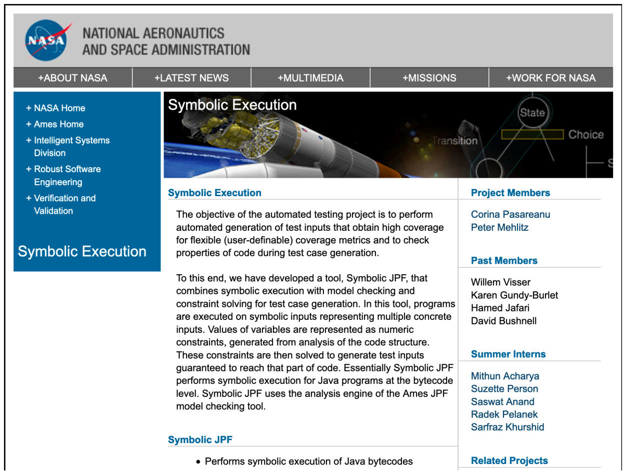
- ✓ Programmer familiarity
- ✓ Concrete input for debugging
- ✓ No false positives
- ✓ Easy regression



Why Automated Testing?



Automated Testing Hits the Mainstream


NATIONAL AERONAUTICS
AND SPACE ADMINISTRATION

+ABOUT NASA | +LATEST NEWS | +MULTIMEDIA | +MISSIONS | +WORK FOR NASA

+ NASA Home
+ Ames Home
+ Intelligent Systems Division
+ Robust Software Engineering
+ Verification and Validation

Symbolic Execution

Symbolic Execution

State
Transition
Choice

Symbolic Execution

The objective of the automated testing project is to perform automated generation of test inputs that obtain high coverage for flexible (user-definable) coverage metrics and to check properties of code during test case generation.

To this end, we have developed a tool, Symbolic JPF, that combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially Symbolic JPF performs symbolic execution for Java programs at the bytecode level. Symbolic JPF uses the analysis engine of the Ames JPF model checking tool.

Symbolic JPF

- Performs symbolic execution of Java bytecodes

Project Members

Corina Pasareanu
Peter Mehlitz

Past Members

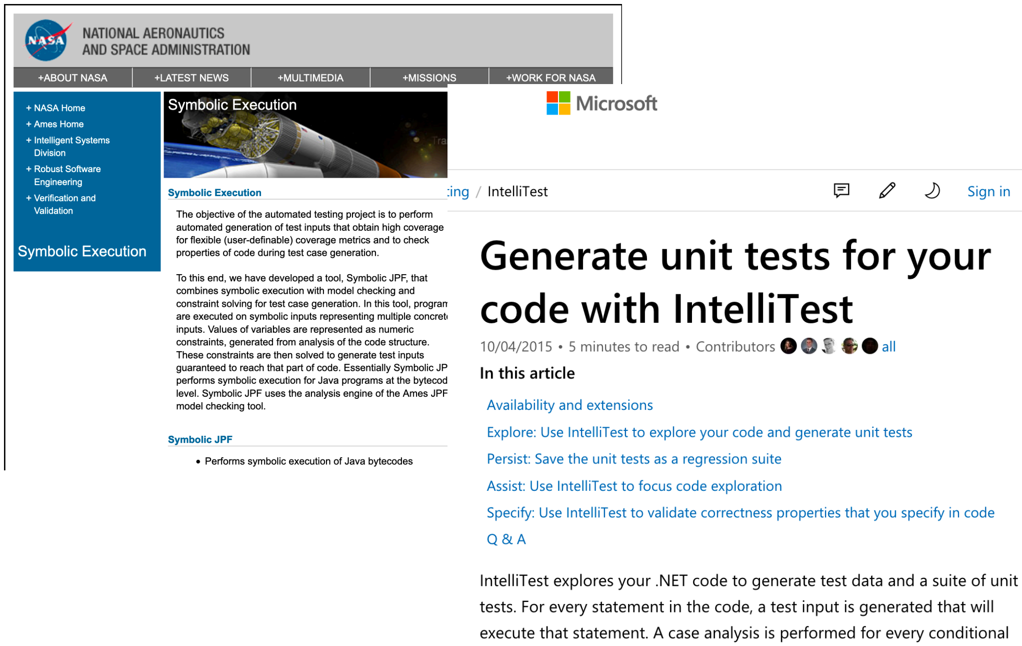
Willem Visser
Karen Gundy-Burlet
Hamed Jafari
David Bushnell

Summer Interns

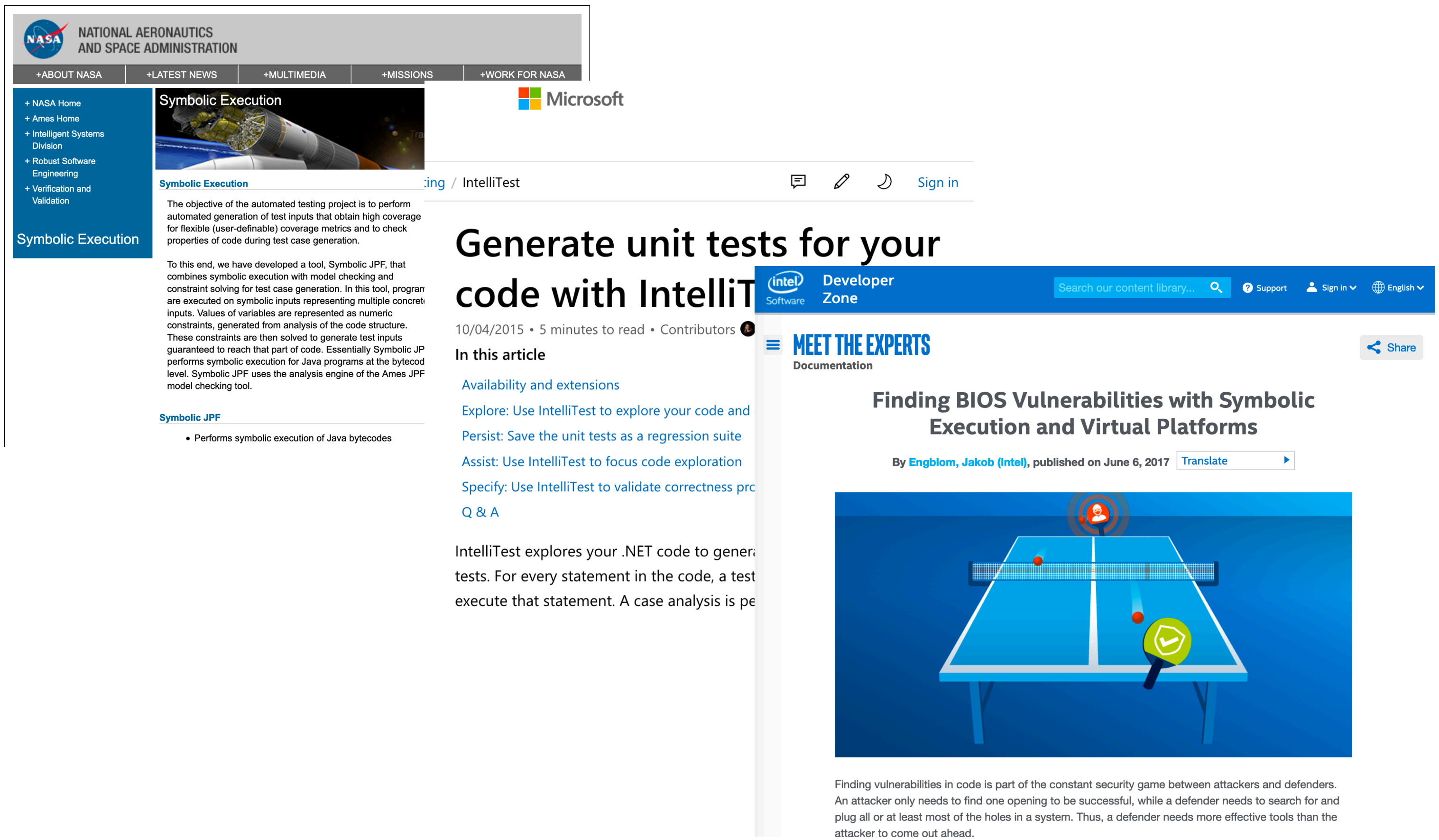
Mithun Acharya
Suzette Person
Saswat Anand
Radek Pelanek
Sarfraz Khurshid

Related Projects

Automated Testing Hits the Mainstream



Automated Testing Hits the Mainstream



NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

+ABOUT NASA | +LATEST NEWS | +MULTIMEDIA | +MISSIONS | +WORK FOR NASA

+ NASA Home
+ Ames Home
+ Intelligent Systems Division
+ Robust Software Engineering
+ Verification and Validation

Symbolic Execution

Symbolic Execution

Symbolic Execution

The objective of the automated testing project is to perform automated generation of test inputs that obtain high coverage for flexible (user-definable) coverage metrics and to check properties of code during test case generation.

To this end, we have developed a tool, Symbolic JPF, that combines symbolic execution with model checking and constraint solving for test case generation. In this tool, program are executed on symbolic inputs representing multiple concret inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially Symbolic JP performs symbolic execution for Java programs at the bytecod level. Symbolic JPF uses the analysis engine of the Ames JPF model checking tool.

Symbolic JPF

- Performs symbolic execution of Java bytecodes

Microsoft

ing / IntelliTest   Sign in

Generate unit tests for your code with IntelliT

10/04/2015 • 5 minutes to read • Contributors

In this article

Availability and extensions
Explore: Use IntelliTest to explore your code and
Persist: Save the unit tests as a regression suite
Assist: Use IntelliTest to focus code exploration
Specify: Use IntelliTest to validate correctness prc
Q & A

IntelliTest explores your .NET code to genera tests. For every statement in the code, a test execute that statement. A case analysis is pe

intel Software   Developer Zone

Search our content library...   Support   Sign in   English

MEET THE EXPERTS
Documentation

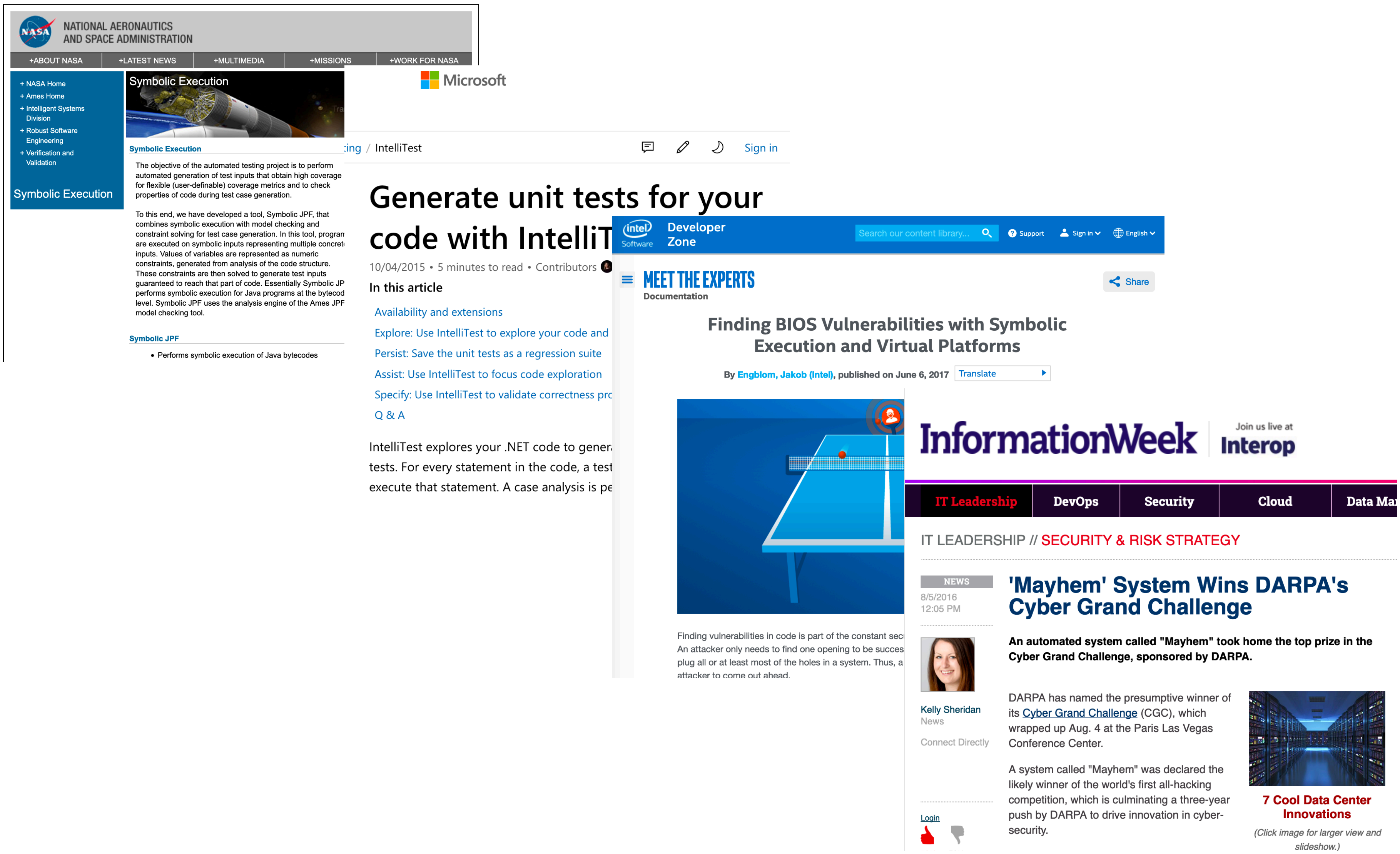
Share

Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms

By Engblom, Jakob (Intel), published on June 6, 2017   Translate

Finding vulnerabilities in code is part of the constant security game between attackers and defenders. An attacker only needs to find one opening to be successful, while a defender needs to search for and plug all or at least most of the holes in a system. Thus, a defender needs more effective tools than the attacker to come out ahead.

Automated Testing Hits the Mainstream



Automated Test Generation Trend

- 1976: King'76, Clarke'76, Howden'77
- 2000: Java PathFinder
- 2001: Started my PhD UIUC
- 2001: SLAM/Blast: Automatic predicate abstraction
- 2001: Java PathExplorer: Runtime Verification
- 2003: Runtime monitoring with Eagle (Internship)
- 2003: Generalized Symbolic Execution

Automated Test Generation Trend

- 1976: King'76, Clarke'76, Howden'77
- 2000: Java PathFinder
- 2001: Started my PhD UIUC
- 2001: SLAM/Blast: Automatic predicate abstraction
- 2001: Java PathExplorer: Runtime Verification
- 2003: Runtime monitoring with Eagle (Internship)
- 2003: Generalized Symbolic Execution
- 2005: DART: Directed Automated Random Testing (Internship)
- 2005: CUTE: A Concolic Unit Testing Engine for C
- 2006: jCUTE: Concolic Testing for Multi-threaded programs

Symbolic JPF, KLEE, CREST, S^2E, Angr, Veritesting, Mayhem, Triton, Jalangi, CATG stackoverflow Products Customers Use cases Search…

Home

PUBLIC

Stack Overflow

Tags

Users

Developer Jobs

TEAMS What's this?

Q&A for Work

In concolic testing, what does “concrete execution” mean?

Viewed 3k times

8

I came across the terms "concrete & symbolic execution" when I was going through the concept of concolic testing. (The article mentioned there, *"CUTE: A concolic unit testing engine for C"*, uses that term in its abstract section.)

> *"The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths."*

Can anyone please confirm what "concrete execution" means? In spite of my search, I could not find any direct citations / explicit statements.

From what I have understood, "concrete execution" means "the execution of a program with actual input values unlike symbolic execution, which assumes symbolic values to variables, inputs etc.". If I am wrong, please correct me (if possible with a small example).

testing execution formal-verification

share improve this question

edited Jan 24 '15 at 18:47 stakx 66.4k ●16 ●139 ●228

asked Jan 24 '15 at 18:16 Suhas Chikkanna 370 ●2 ●6 ●22

add a comment

2 Answers

active oldest votes

14

Concolic execution is a mix between CONCrete execution and symbOLIC execution, with the purpose of feasibility.

Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions (path condition = the set of logical constraints that takes us to a

Quora Search for questions, people, and topics

Quora uses cookies to improve your experience. Read more

Concolic Testing Automated Testing Software Testing

What is concolic testing?

1 Answer

Martin Gollogly, Develop Test Frameworks, Usually write and test code
Answered Oct 26 2016 · Author has **205** answers and **418.5k** answer views

Concolic is a portmanteau of **CONC**rete execution and symb**OLIC** execution. It’s also known as Directed Automated Random Testing (DART). It’s a way to automatically and systematically generate test inputs for programs.

A good definition can be found at Concolic testing - Wikipedia which also has many useful links for further reading

422 views · View 1 Upvoter

Related Questions

What is Concolic testing?

- Combine concrete execution and symbolic execution

Concrete + Symb**olic** = Concolic

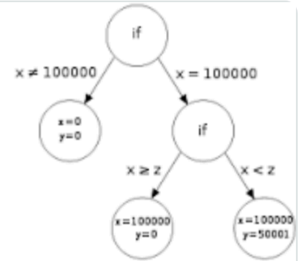
Google

concolic

All · Images · News · Videos · Maps · More · Settings · Tools

About 172,000 results (0.34 seconds)

Concolic testing (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path.



Concolic testing - Wikipedia

https://en.wikipedia.org › wiki › Concolic_testing

About Featured Snippets · Feedback

Concolic testing - Wikipedia

https://en.wikipedia.org › wiki › Concolic_testing

Concolic testing (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path.

Birth of concolic testing · Example · Algorithm · Commercial success

Concolic Fuzzing - Generating Software Tests - Fuzzing Book

https://www.fuzzingbook.org › html › ConcolicFuzzer

The idea of **concolic** execution over a function is as follows: We start with a sample input for the function, and execute the function under trace. At each point the ...

[PDF] Symbolic and Concolic Execution - Verimag

www-verimag.imag.fr › ~mounier › Enseignement › Software_Security

Each theory comes with a set of axioms (FOL formulas), called A, which only contain elements from the signature. The predicates and functions in have no ...

why on earth is concolic execution better? · Issue #907 · klee/kle...

Goal

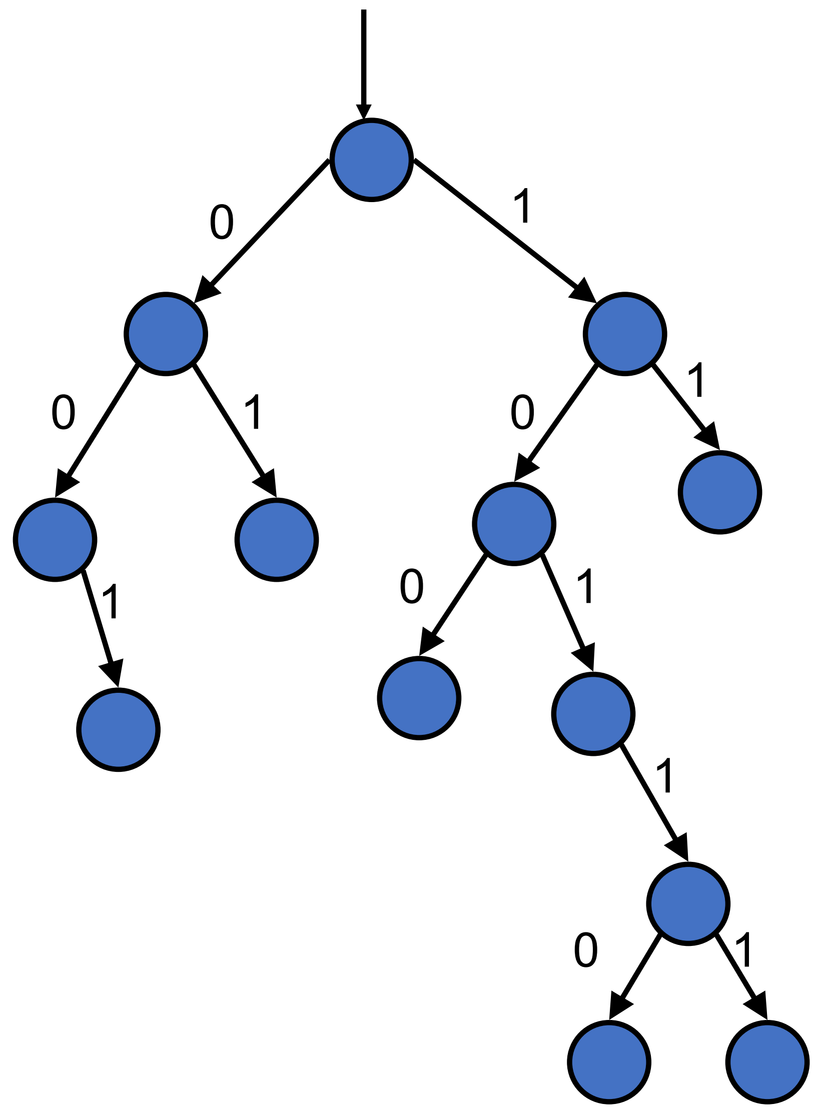
- Automated Unit Testing of real-world C and Java Programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught

Goal

- Automated Unit Testing of real-world C and Java Programs
  - Generate test inputs
  - Execute unit under test on generated test inputs
    - so that all reachable statements are executed
  - Any assertion violation gets caught
- Concolic Testing Approach:
  - Explore all execution paths of an unit for all possible inputs

Computation Tree

- Can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of a "if then else" statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

            if (x > y+10) {

                ERROR;
            }
    }

}
```

- Random Test Driver:
  - random values for x and y
- Probability of reaching ERROR is extremely low

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 22, y = 7$	$x = x_0, y = y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
$x = 22, y = 7, z = 14$	$x = x_0, y = y_0, z = 2^*y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
			$2*y_0 \mathrel{!}= x_0$
	$x = 22, y = 7, z = 14$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
$x = 22,\ y = 7,\ z = 14$	$x = x_0,\ y = y_0,\ z = 2^*y_0$	$2^*y_0 \mathrel{!=} x_0$

Solve: $2^*y_0 == x_0$

Solution: $x_0 = 2,\ y_0 = 1$

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

         if (x > y+10) {

              ERROR;
         }
    }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 2, y = 1$	$x = x_0, y = y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

concrete state	symbolic state	path condition
Concrete Execution	Symbolic Execution	
$x = 2$, $y = 1$, $z = 2$	$x = x_0$, $y = y_0$, $z = 2^*y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

$2*y_0 == x_0$

x = 2, y = 1, z = 2

$x = x_0,\ y = y_0,\ z = 2*y_0$

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

            if (x > y+10) {

                ERROR;
            }
    }

}
```

	Concrete Execution		Symbolic Execution
concrete state		symbolic state	path condition
			$2^*y_0 == x_0$
			$x_0 \leq y_0+10$
$x = 2,\ y = 1,\ z = 2$		$x = x_0,\ y = y_0,\ z = 2^*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition	

Solve: $(2^*y_0 == x_0) \wedge (x_0 > y_0 + 10)$

Solution: $x_0 = 30$, $y_0 = 15$

Path condition: $2^*y_0 == x_0$, $x_0 > y_0+10$

Concrete state: $x = 2$, $y = 1$, $z = 2$

Symbolic state: $x = x_0$, $y = y_0$, $z = 2^*y_0$

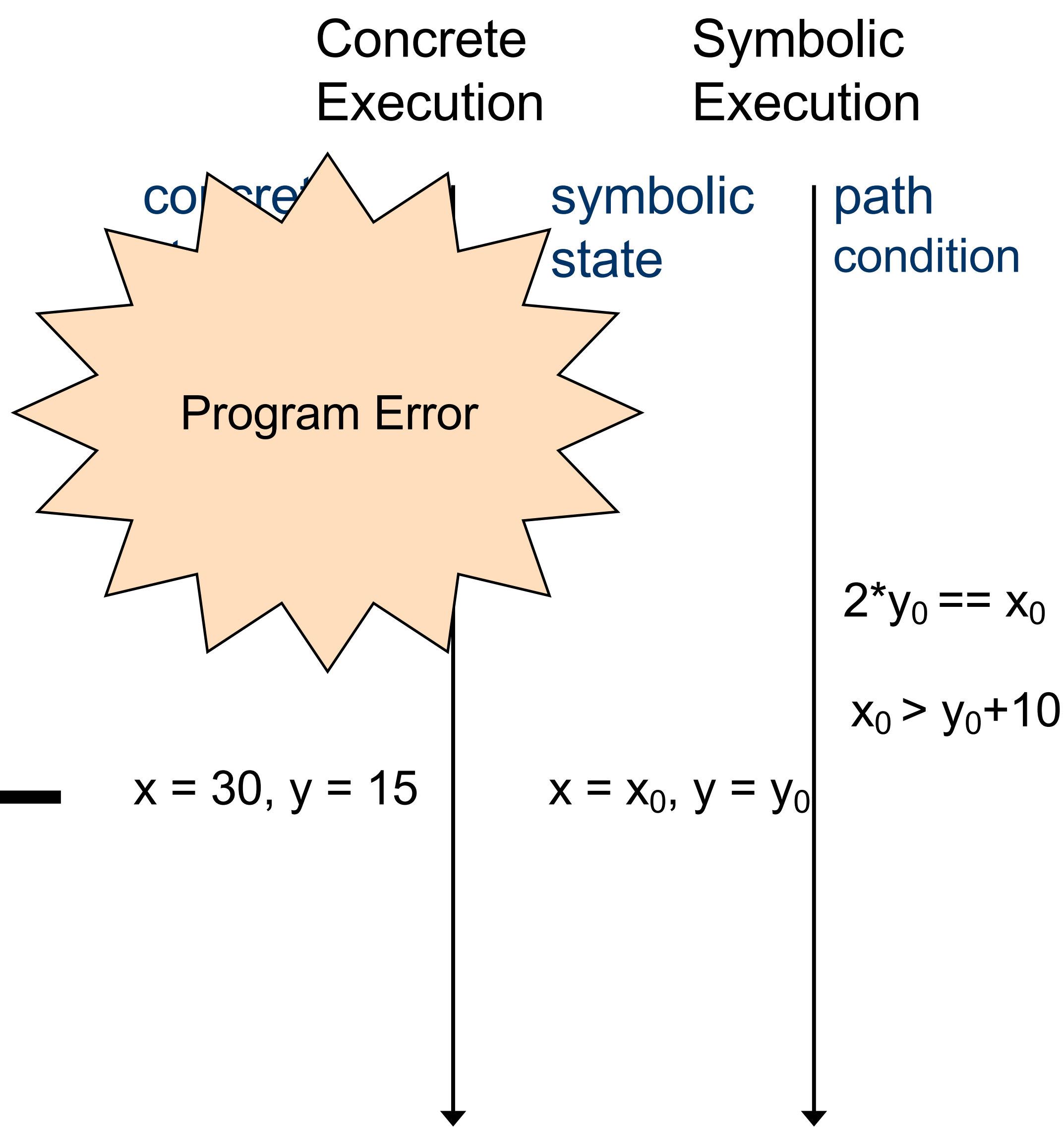
Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

            if (x > y+10) {

                ERROR;
            }
    }

}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 30, y = 15	$x = x_0$, $y = y_0$	

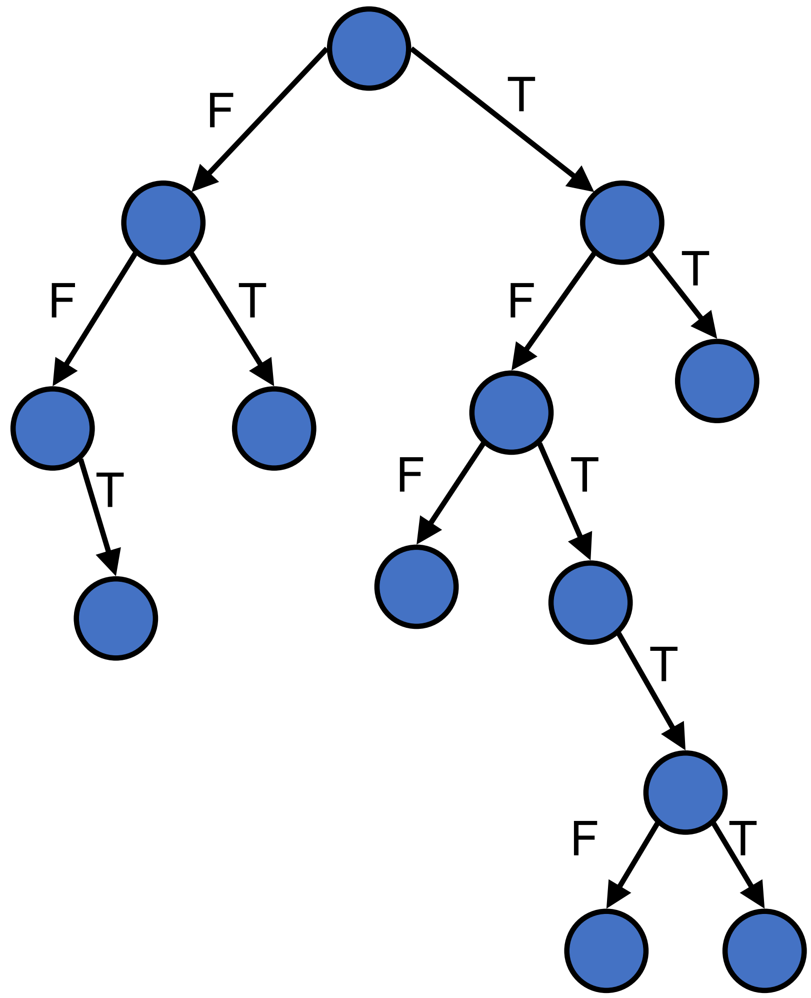
Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

            if (x > y+10) {

                ERROR;

            }
    }

}
```



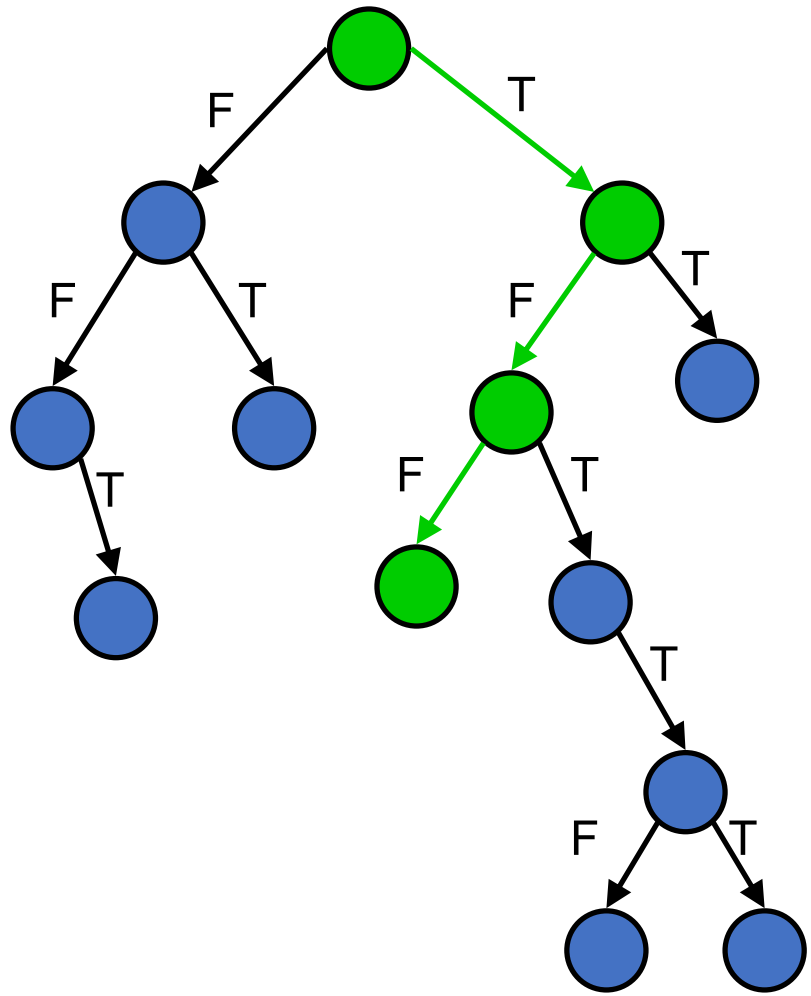
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



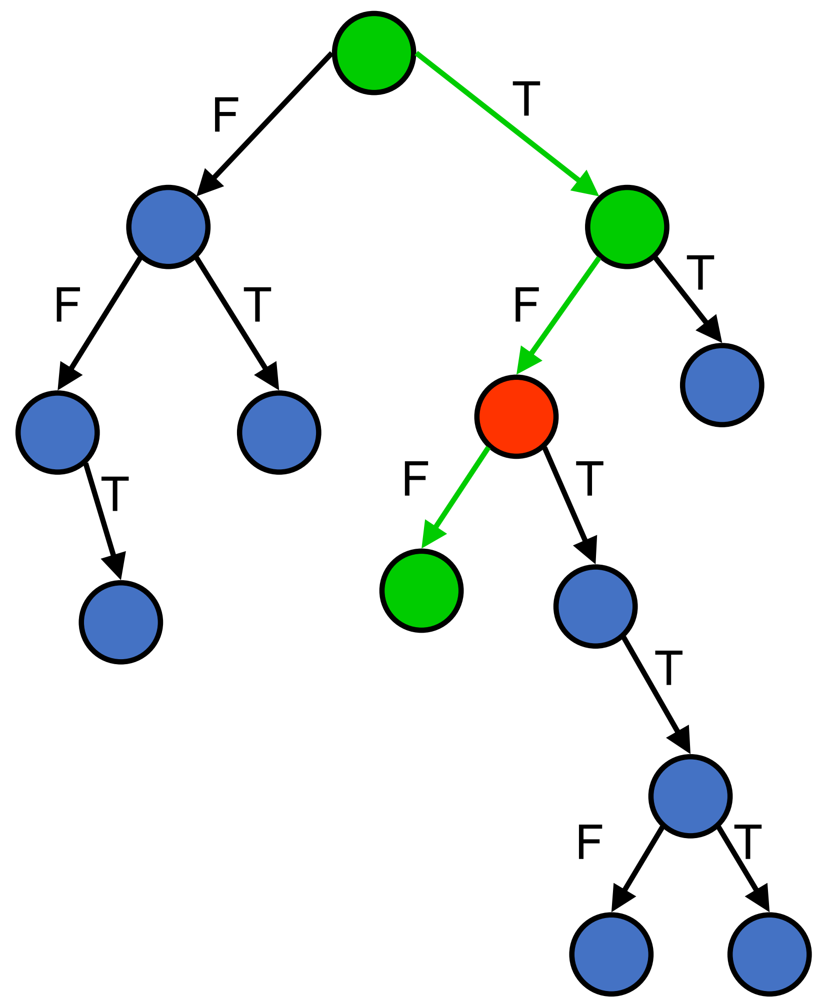
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



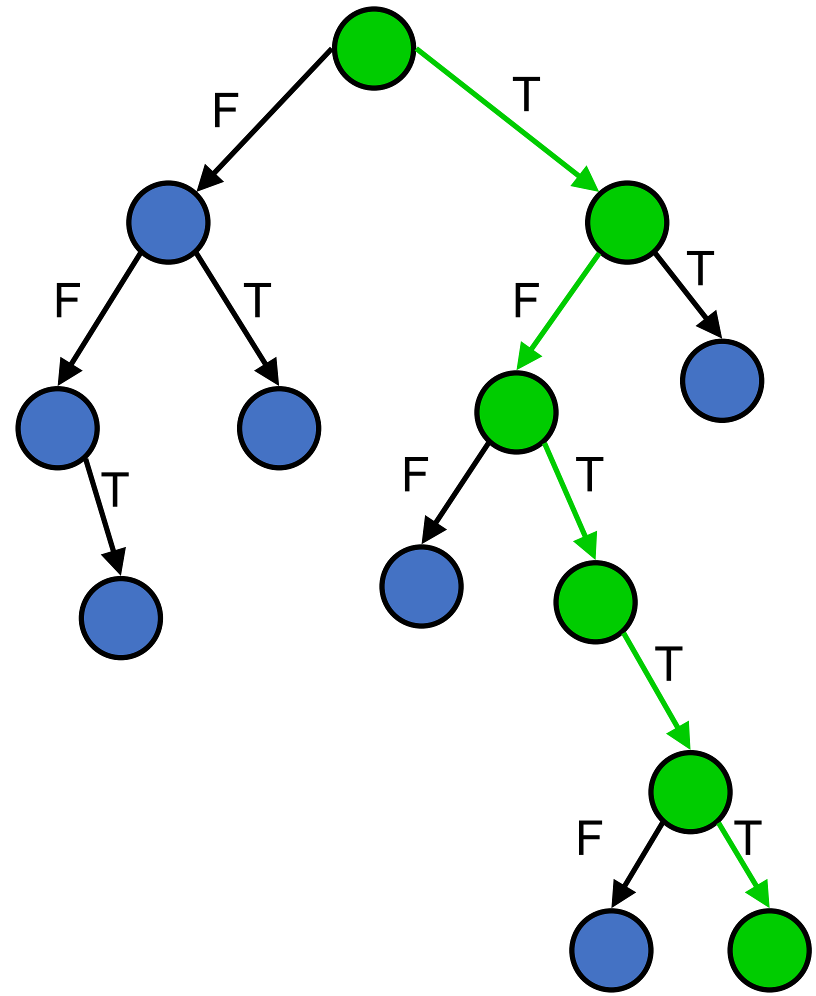
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



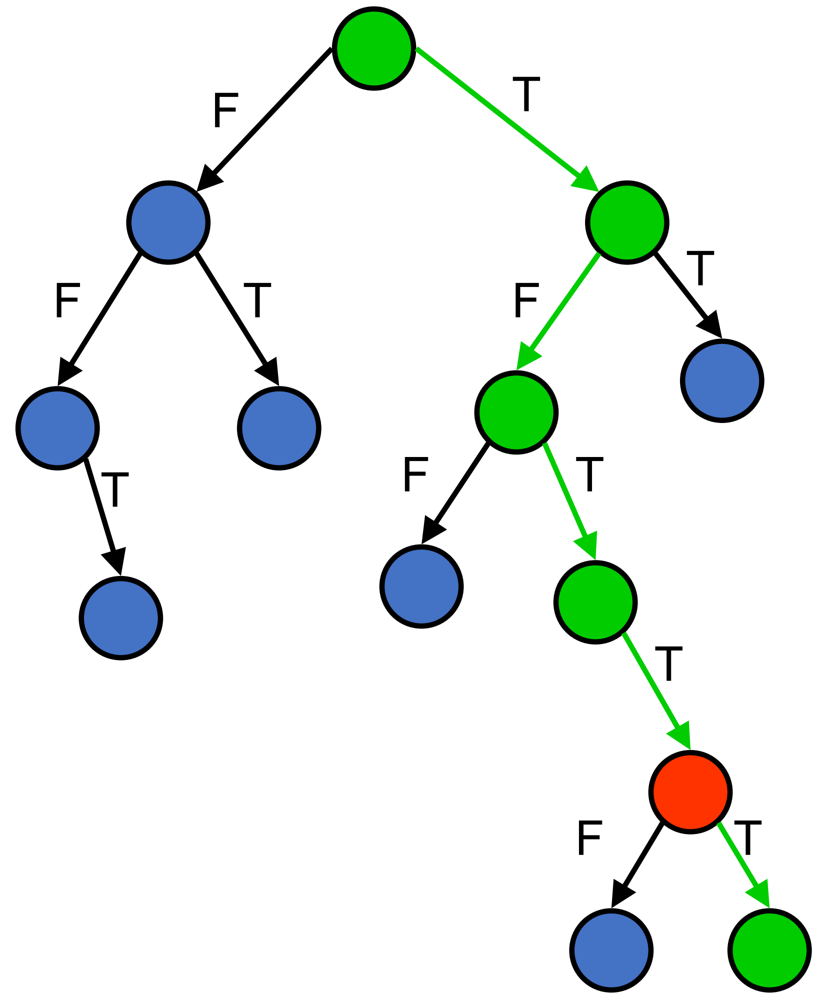
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



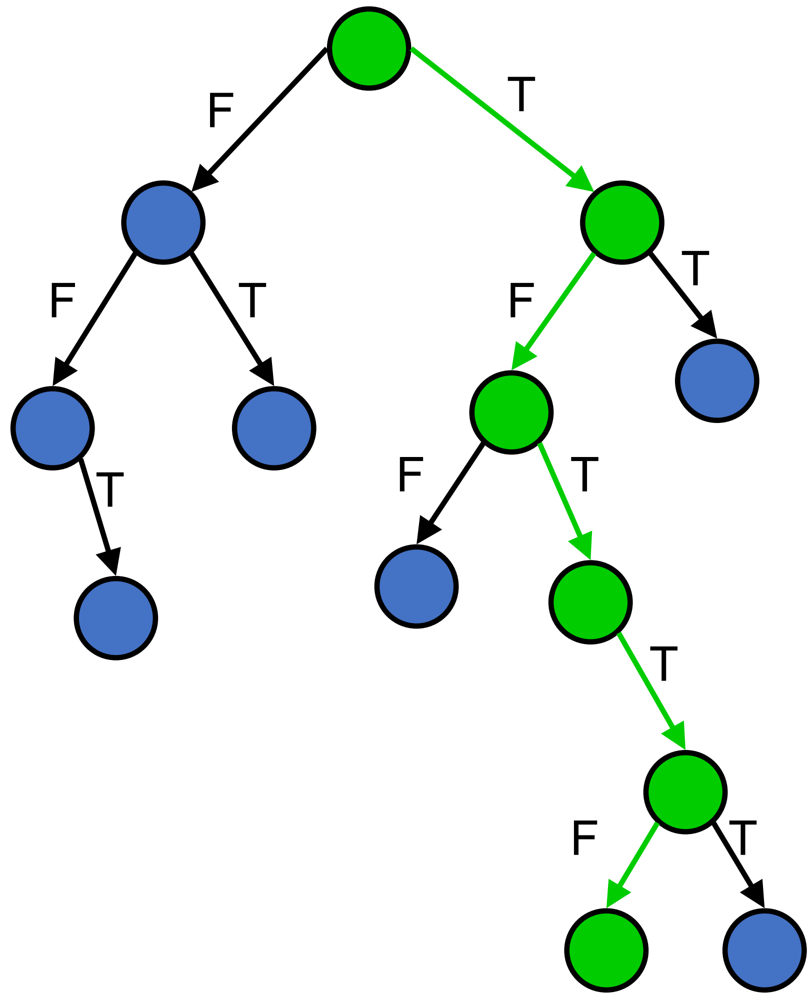
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with address sanitizer to discover memory errors



Novelty : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 22, y = 7$	$x = x_0, y = y_0$	

Novelty : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: $(y_0 * y_0)\%50 == x_0$

Don't know how to solve!

Stuck?

$(y_0 * y_0)\%50 \mathrel{!}= x_0$

$x = 22, y = 7, z = 49$

$x = x_0, y = y_0, z = (y_0 * y_0)\%50$

Novelty : Simultaneous Concrete and Symbolic Execution

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
Solve: foo (y_0) == x_0		
Don't know how to solve!		
Stuck?		foo (y_0) !=x_0
x = 22, y = 7, z = 49	$x = x_0$, $y = y_0$, $z = foo(y_0)$	

```
void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Novelty : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$(y_0*y_0)\%50 \mathrel{!}= x_0$
$x = 22, y = 7, z = 49$	$x = x_0, y = y_0, z = (y_0 * y_0)\%50$	

Solve: $(y_0*y_0)\%50 == x_0$

Don't know how to solve!

Not Stuck!

Use concrete state

Replace y_0 by 7 (sound)

Novelty : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: $49 == x_0$

Solution : $x_0 = 49$, $y_0 = 7$

$49 \,!= x_0$

$x = 22$, $y = 7$, $z = 48$

$x = x_0$, $y = y_0$, $z = 49$

Novelty : Simultaneous Concrete and Symbolic Execution

```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 49, y = 7$	$x = x_0, y = y_0$	

Novelty : Simultaneous Concrete and Symbolic Execution

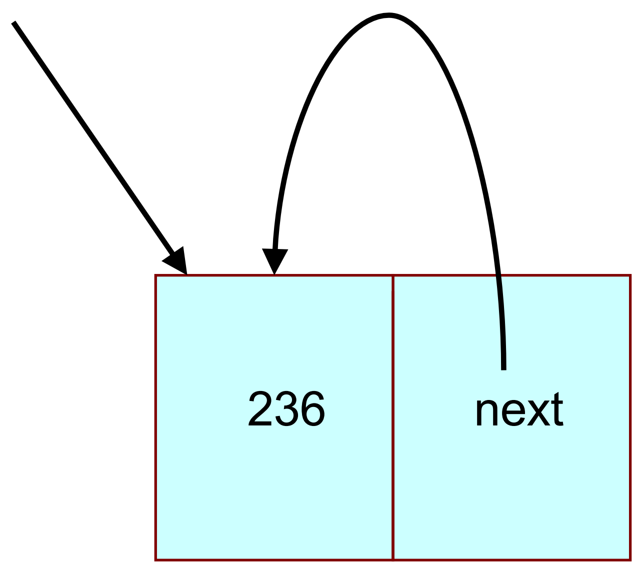
```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 > y_0+10$
$x = 49, y = 7, z = 49$	$x = x_0, y = y_0, z = 49$	

Program Error

Summary: Pointers and Data-Structures

Logical Input Map to symbolically represent Memory Graph pointed by an input Pointer



{0 → 1, 1 → 236, 2 → 1 }

- Pointer Constraints
  - p ≠ NULL
  - p = NULL
  - p ≠ q
  - p = q
- Solving Pointer Constraints
  - Construct equivalence class [p] for each pointer input p
  - p ≠ NULL

    Add a node and point [p] to it
  - p = NULL

    Delete node pointed by [p]
  - p = q

    Make [p] and [q] point to same node
  - p ≠ q

    Add a node and point [p] or [q] to it

Concolic Testing: Finding Security and Safety Bugs

<u>Divide by 0 Error</u>

```
x = 3 / i;
```

<u>Buffer Overflow</u>

```
a[i] = 4;
```

Concolic Testing: Finding Security and Safety Bugs

Key: Add Checks Automatically and

Perform Concolic Testing

<u>Divide by 0 Error</u>

```
if (i !=0)
   x = 3 / i;
else
   ERROR;
```

<u>Buffer Overflow</u>

```
if (0<=i  &&  i < a.length)
   a[i] = 4;
else
   ERROR;
```

Incremental Constraint Solving

- Observation: one constraint is negated at each execution
  - $C1 \land C2 \land \ldots \land Ck$ has a satisfying assignment
  - Need to solve $C1 \land C2 \land \ldots \land \neg Ck$
  - Previous solution more or less similar to current solution
  - Eliminate non-dependent constraints

$$(x==1) \land (y>2) \land \neg (y==4)$$

to

$$(y>2) \land \neg (y==4)$$

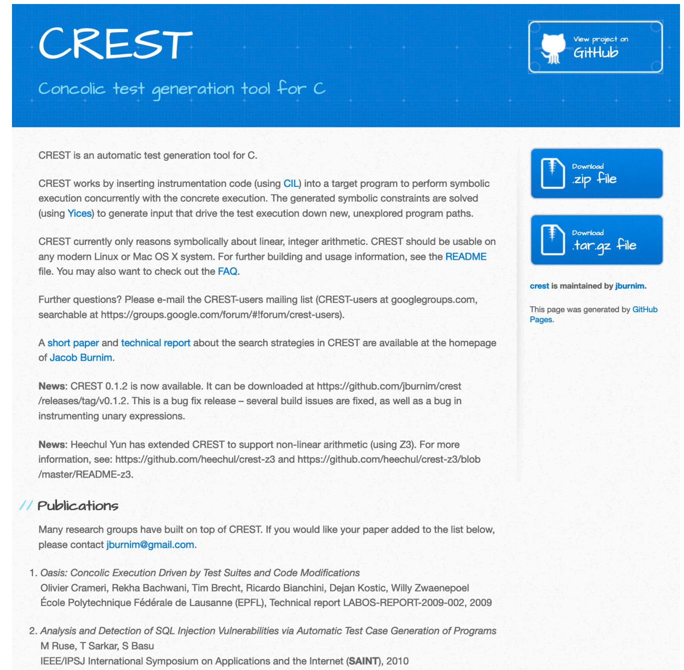
- Incremental Solving
  - 100 -1000 times faster than a naïve solver

Underlying Random Testing Helps

```
1  foobar(int x, int y){
2    if (x*x*x > 0){
3       if (x>0 && y==10){
4           ERROR;
5       }
6    } else {
7       if (x>0 && y==20){
8           ERROR;
9       }
10   }
11 }
```

- static analysis based model-checkers would consider both branches
    - both ERROR statements are reachable
    - false alarm
- Symbolic execution
    - gets stuck at line number 2
    - or warn that both ERRORs are reachable
- CUTE finds the only error

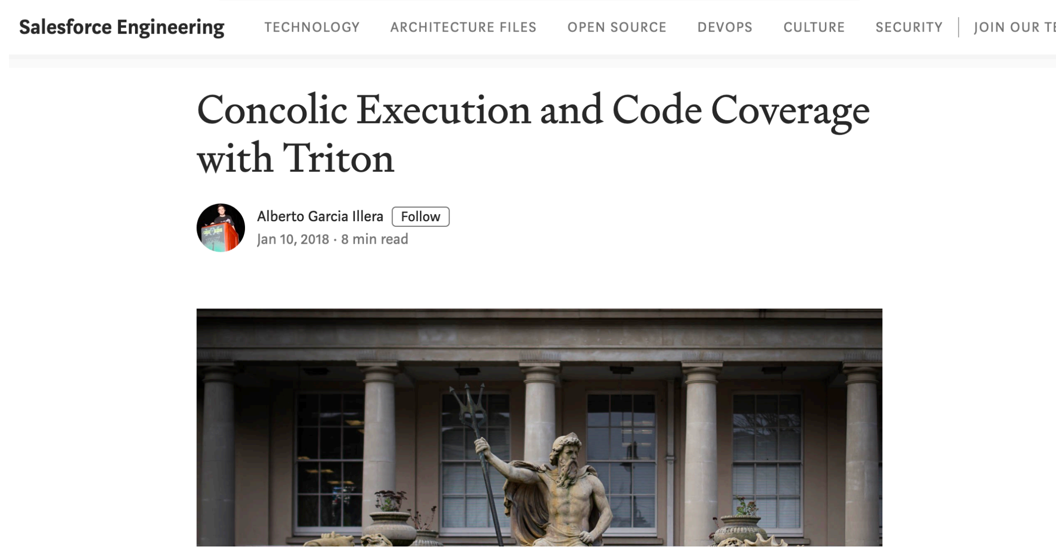
DART, CUTE, jCUTE, CREST, Jalangi, CATG



- DART for C
- CUTE for C and jCUTE for Java
  - 5000+ downloads (around 2010)
  - used in both academia and industry
- CREST
  - extensible open-source tool for C
- Jalangi for JavaScript Concolic Testing
- CATG for Concolic Testing of Java bytecode
  - https://github.com/ksen007/janala2

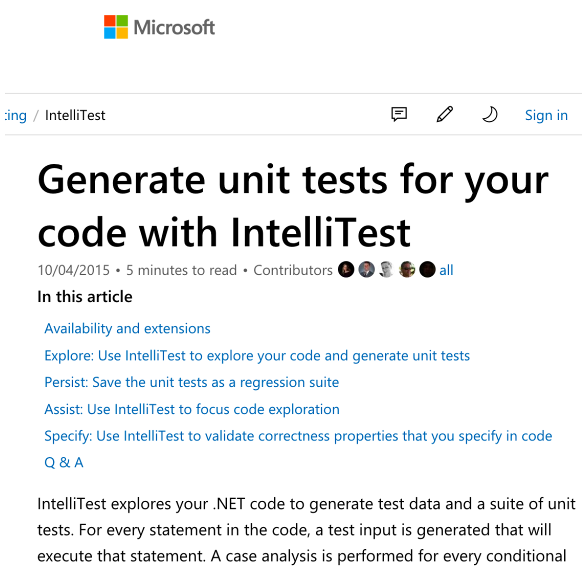
Concolic Testing in Practice

- Led to the development of several industrial and academic automated testing and security tools
  - Projects at Intel, Google, MathWorks, NTT, SalesForce
  - PEX, SAGE, and YOGI at Microsoft
  - Apollo at IBM, and Conbol and Jalangi at Samsung
  - BitBlaze, jFuzz, Oasis, and SmartFuzz in academia



Concolic Testing in Practice

- Led to the development of several industrial and academic automated testing and security tools
  - Projects at Intel, Google, MathWorks, NTT, SalesForce
  - PEX, SAGE, and YOGI at Microsoft
  - Apollo at IBM, and Conbol and Jalangi at Samsung
  - BitBlaze, jFuzz, Oasis, and SmartFuzz in academia


Microsoft

ing / IntelliTest Sign in

Generate unit tests for your code with IntelliTest

10/04/2015 • 5 minutes to read • Contributors all

In this article

Availability and extensions
Explore: Use IntelliTest to explore your code and generate unit tests
Persist: Save the unit tests as a regression suite
Assist: Use IntelliTest to focus code exploration
Specify: Use IntelliTest to validate correctness properties that you specify in code
Q & A

IntelliTest explores your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional


Microsoft | Research Research areas Researcher tools More All Microsoft

QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, Taesoo Kim
27th USENIX Security Symposium (Security 2018) | August 2018
Distinguished Paper Award Winner
Download BibTex

Many Applications

ars TECHNICA SUBSCRIBE

CLOUDY, WITH A CHANCE OF EXPLOITS —

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft's "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Many Applications

ars TECHNICA SUBSCRIBE

CLOUDY, WITH A CHANCE OF EXPLOITS —

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft's "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

POST APRIL 1, 2019 1 COMMENT

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Many Applications

ars TECHNICA SUBSCRIBE

CLOUDY, WITH A CHANCE OF EXPLOITS —

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft's "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

POST APRIL 1, 2019 1 COMMENT

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Con2colic testing

Full Text: PDF Get this Article

Authors: Azadeh Farzan University of Toronto, Canada
Andreas Holzer Vienna University of Technology, Austria
Niloofar Razavi University of Toronto, Canada
Helmut Veith Vienna University of Technology, Austria

Institutional Profile Page

Published in:
· Proceeding
ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on F
Pages 37-47

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li ; Sihuan Li ; Zachary Benavides ; Zizhong Chen ; Rajiv Gupta **View All Authors**

Many Applications

ars TECHNICA SUBSCRIBE

CLOUDY, WITH A CHANCE OF EXPLOITS —

Microsoft launches "fuzzing-as-a-service" to help developers find security bugs

Project Springfield, Microsoft's "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

POST APRIL 1, 2019 1 COMMENT

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

DeepConcolic (Concolic Testing for Deep Neural Networks)

DeepConcolic
Safety for AI

Con2colic testing

Full Text: PDF Get this Article

Authors:
Azadeh Farzan University of Toronto, Canada
Andreas Holzer Vienna University of Technology, Austria
Niloofar Razavi University of Toronto, Canada
Helmut Veith Vienna University of Technology, Austria

Institutional Profile Page

Published in:
· Proceeding
ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on F
Pages 37-47

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li ; Sihuan Li ; Zachary Benavides ; Zizhong Chen ; Rajiv Gupta **View All Authors**

Many Applications

ars TECHNICA SUBSCRIBE

CLOUDY, WITH A CHANCE OF EXPLOITS —

Microsoft launches "fuzzing-as-a-service" to help developers find security bugs

Project Springfield, Microsoft's "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

POST APRIL 1, 2019 1 COMMENT

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Con2colic testing

Full Text: PDF Get this Article

Authors: Azadeh Farzan University of Toronto, Canada
Andreas Holzer Vienna University of Technology, Austria
Niloofar Razavi University of Toronto, Canada
Helmut Veith Vienna University of Technology, Austria

Institutional Profile Page

Published in:
· Proceeding
ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on F
Pages 37-47

DeepConcolic (Concolic Testing for Deep Neural Networks)

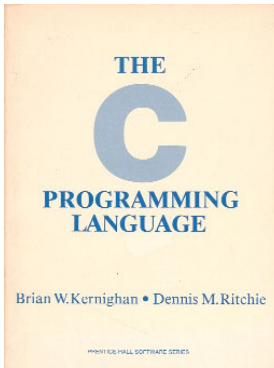
DeepConcolic
Safety for AI

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li ; Sihuan Li ; Zachary Benavides ; Zizhong Chen ; Rajiv Gupta **View All Authors**

Many Languages



C++

Java

JavaScript

JS

python™

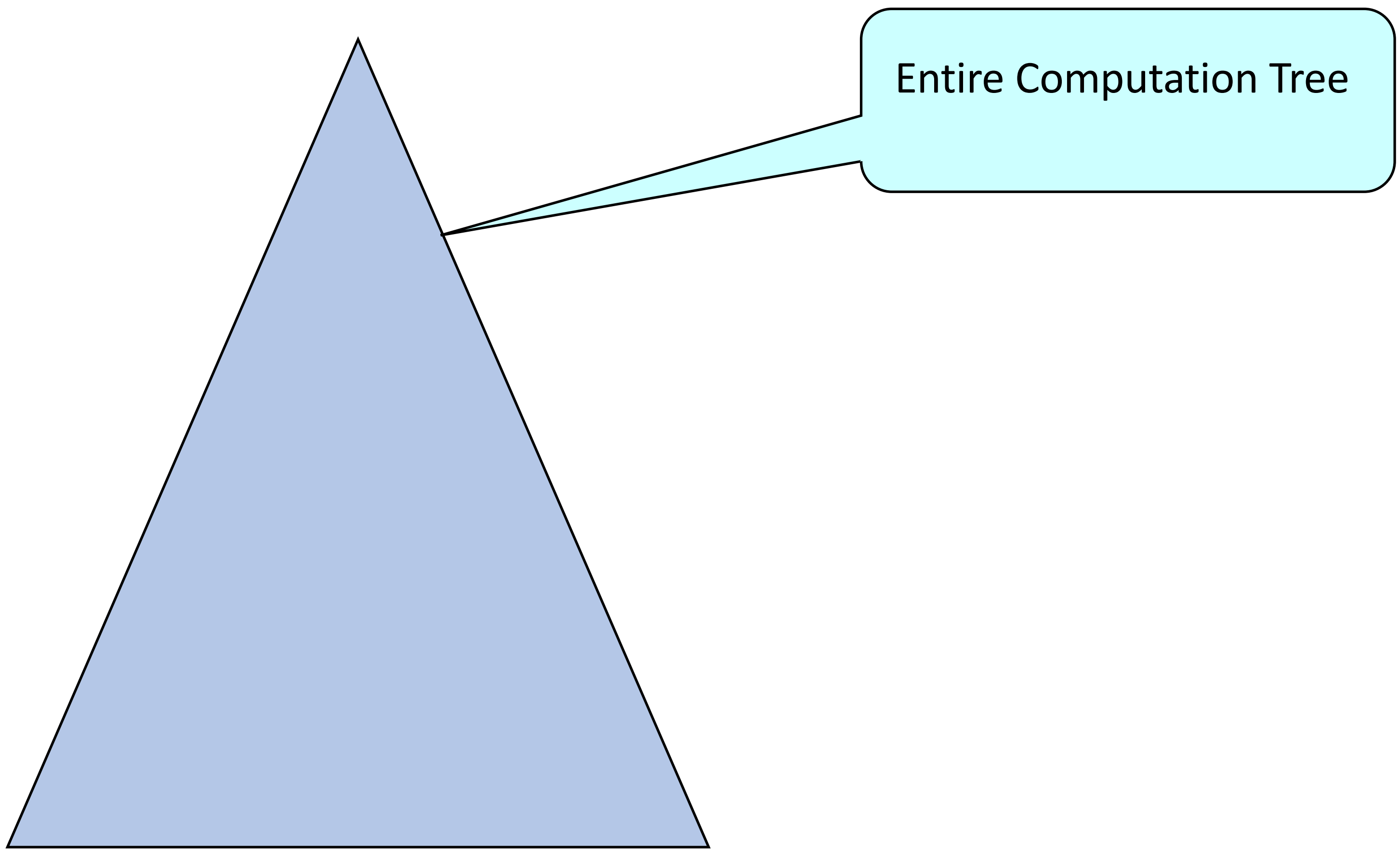
LLVM

php

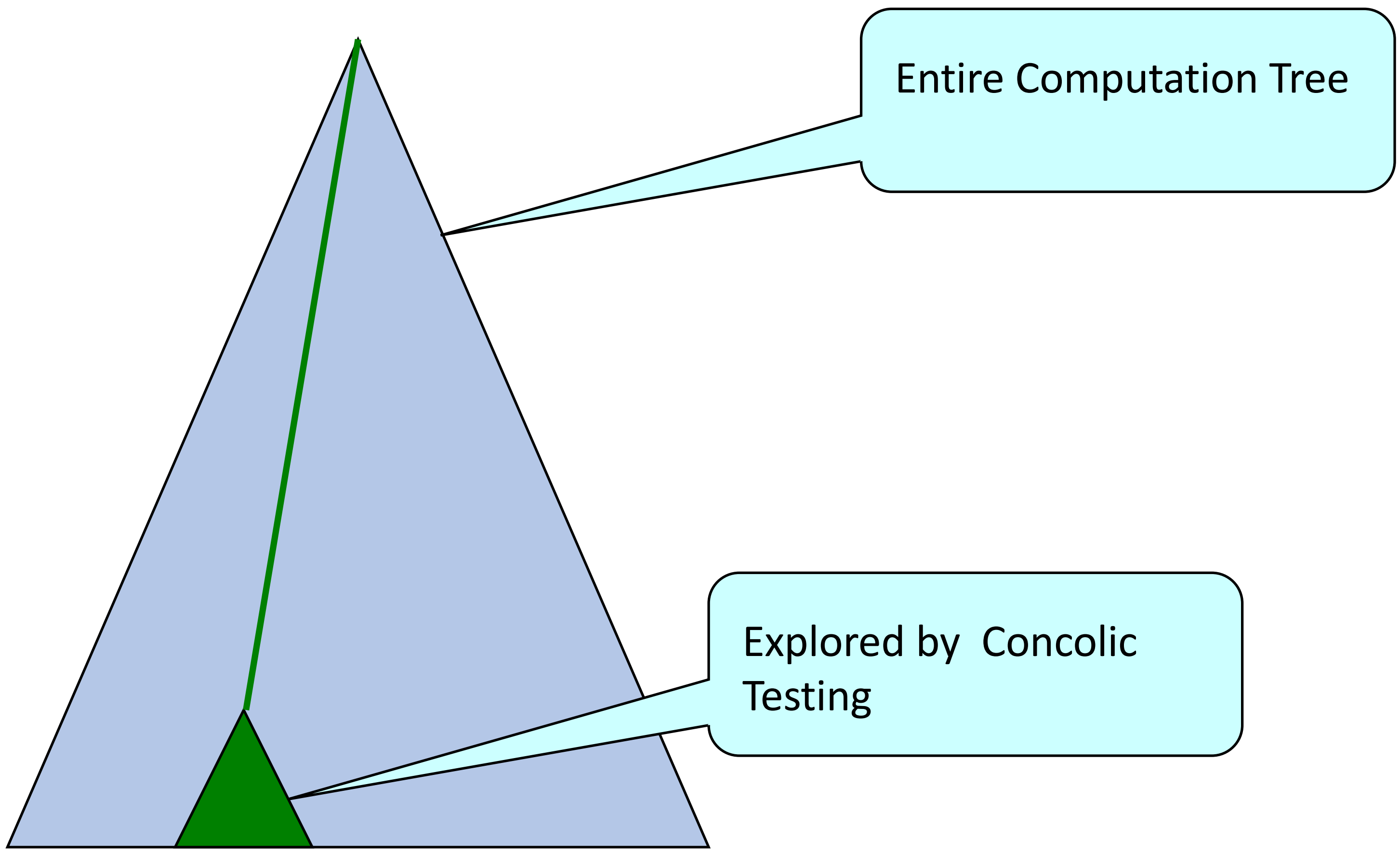
x86

Java bytecode

Concolic Testing: Path-explosion Problem



Concolic Testing: Path-explosion Problem



Scaling Concolic Testing

- Control-flow Directed Search (CREST)
- Combining fuzzing and concolic testing (Hybrid Concolic Testing, Driller, Mayhem)
- Function Summaries (SMART, Veritesting)
- Loop Summaries (Proteus, LESE)
- State Merging using Value Summaries (MultiSE)
- Interpolation (Tracer)
- Abstract Subsumption Checking
- Pruning redundant paths (RWSet)
- Parallel techniques (Siddiqui & Khurshid, and Staats & Pasareanu)
- Incremental techniques (Person et al.)
- …

Lessons Learned

- Focused on an important real-world problem
- Did not try to invent from the beginning
  - Tried existing approaches to solve a real problem
  - Observed limitations
  - Got insights → led to effective solutions
  - Identified novel contributions (and wrote papers)

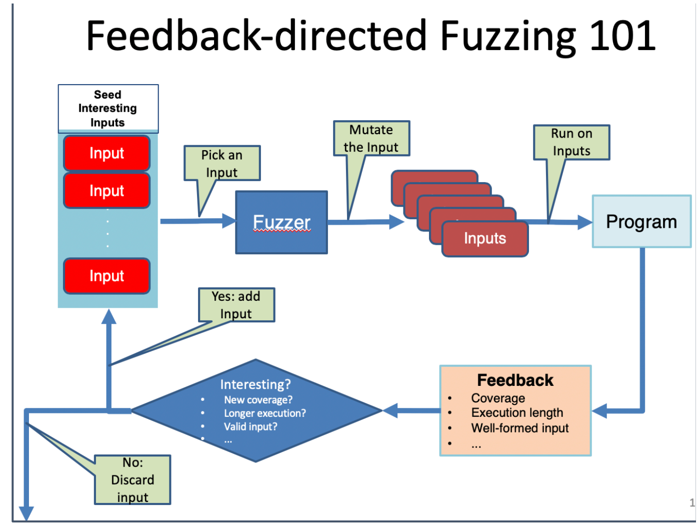
Things We Should Have Done Differently

- If there is a big idea for a practical problem
  - Build a practical system that users can use
  - Promote the area of research
    - Your competitors are your real-friends
  - Do not hesitate to use competing techniques
    - If it helps to solve the problem
  - Take feedback seriously
    - From actual users
    - And reviewers

What we do now

- We target real-world problems
- We target real software in the more popular languages
  - rather than assuming a nice clean slate for research
  - leads us to see a lot of problems
- We build prototypes before building a large system
- We release our tools as open-source software
  - so that the tools are usable by the broader community
- We release our benchmarks

Smart Fuzzing



Intention

- Correctness **FairFuzz**
- Performance Bugs **PerfFuzz**
- Custom testing Goals **FuzzFactory**
- Semantic Fuzzing (**Zest)**
- Constraint Fuzzing **QuickSampler** **SMTSampler (Dutra)**

Algorithm

- Symbolic Execution/ Concolic Testing **CUTE**
- Genetic algorithm **AFL**
- Reinforcement Learning **RLCheck**
- Neural Network ???
- Bayesian Learning ???

Implementation

- Parallelization
- LLVM, x86
- Java Virtual Machine **JQF, RLCheck**
- Python **RLCheck**
- RTL using FPGA **RFuzz (Laeufer)**

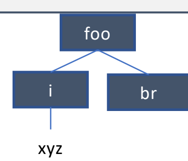
Zest: Semantic Fuzzing

Padhye, Lemieux, Sen, Papadakis, Le Traon

```java
public XMLElement genXML(Random random) {
  // Generate a random tag name
  String name = random.nextString(MAX_TAG_LENGTH);
  XMLElement node = new XMLElement(name);

  // Generate a random number of children
  int n = random.nextInt(MAX_CHILDREN);
  for (int i = 0; i < n; i++) {
    // Generate child nodes recursively
    node.addChild(genXML(random));
  }
  // Maybe insert text inside element
  if (random.nextBoolean()) {
    node.addText(random.nextString(MAX_TEXT_LENGTH));
  }
  return node;
}
```

- Developer writes a simple input generator as a program
- Generator restricts the space of inputs

Example generated:



<foo><i>xyz</i>
</foo>

Zest: New bugs discovered

- ✓ **Google Closure Compiler**: #2842, #2843, #3220, #3173
- ✓ **OpenJDK**: JDK-8190332, JDK-8190511, JDK-8190512, JDK-8190997, JDK-8191023, JDK-8191076, JDK-8191109, JDK-8191174,JDK-8191073, JDK-8193444, JDK-8193877, **CVE-2018-3214**
- ✓ **Apache Commons**: LANG-1385, COMPRESS-424, COLLECTIONS-714, **CVE-2018-11771**
- ✓ **Apache Ant**: #62655
- ✓ **Apache Maven**: #34, #57
- ✓ **Apache PDFBox**: PDFBOX-4333, PDFBOX-4338, PDFBOX-4339, **CVE-2018-8036**
- ✓ **Apache TIKA**: **CVE-2018-8017**, **CVE-2018-12418**
- ✓ **Apache BCEL**: BCEL-303, BCEL-307, BCEL-308, BCEL-309, BCEL-310, BCEL-311, BCEL-312, BCEL-313
- ✓ **Mozilla Rhino**: #405, #406, #407, #409, #410

QuickSampler, SMTSampler, GuidedSampler Human Writes a Pre-condition on Inputs

(node.left != NULL => node.val > node.left.val)
/\ (node.right != NULL =>node.val <= node.right.val)

✓ An over-approximation of valid inputs

✓ Restricts the set of inputs to be generated

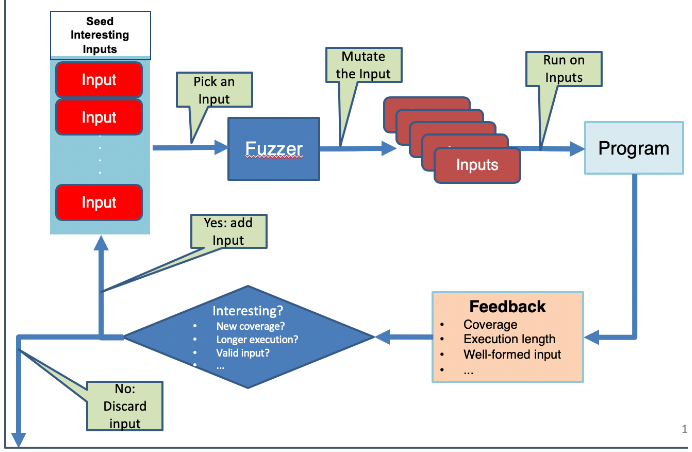
Goal: sample inputs from the restricted input space

Generates more **diverse** set of solutions compared to UniGen2 and SearchTreeSampler

- QuickSampler generates valid solutions
  - $10^{2.5\pm0.8}$ times **faster** than SearchTreeSampler
  - $10^{4.7\pm1.0}$ times **faster** than UniGen2
- QuickSampler generates *unique* valid solutions
  - $10^{2.3\pm0.7}$ times **faster** than SearchTreeSampler
  - $10^{4.4\pm1.1}$ times **faster** than UniGen2

Smart Fuzzing

Feedback-directed Fuzzing 101



Intention	Algorithm	Implementation
Correctness **FairFuzz**	Symbolic Execution/ Concolic Testing **CUTE**	Parallelization
Performance Bugs **PerfFuzz**	Genetic algorithm **AFL**	LLVM, x86
Custom testing Goals **FuzzFactory**	Reinforcement Learning **RLCheck**	Java Virtual Machine **JQF, RLCheck**
Semantic Fuzzing (**Zest**)	Neural Network ???	Python **RLCheck**
Constraint Fuzzing **QuickSampler** **SMTSampler (Dutra)**	Bayesian Learning ???	RTL using FPGA **RFuzz (Laeufer)**

Koushik Sen, Darko Marinov, Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
{ksen,marinov,agha}@cs.uiuc.edu

ABSTRACT

In *unit testing*, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are *memory graphs*. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining *symbolic* and *concrete execution*, and more specifically, using such a combination to *generate test inputs* to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of such test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability,Verification

Keywords: concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs.

1. INTRODUCTION

Unit testing is a method for modular testing of a programs' functional behavior. A program is decomposed into units, where each unit is a collection of functions, and the units are independently tested. Such testing requires specification of values for the inputs (or *test inputs*) to the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or *test coverage*), several techniques have been proposed to automatically generate values for the inputs. One such technique is to *randomly* choose the values over the domain of potential inputs [4,8,10,21]. The problem with such random testing is two fold: first, many sets of values may lead to the same observable behavior and are thus *redundant*, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [20].

One approach which addresses the problem of redundant executions and increases test coverage is *symbolic execution* [1, 3, 9, 22, 23, 27, 28, 30]. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an *execution path*. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking [14]. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To the best of our knowledge, Larson and Austin were the first to propose combining concrete and symbolic execution [16]. In their approach, the program is executed on some user-provided concrete input values. Symbolic path constraints are generated for the specific execution. These constraints are solved, if feasible, to see whether there are potential input values that would have led to a violation along the same execution path. This improves coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths.

Godefroid et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [11]. In Godefroid et al.'s approach, during a concrete execution, a conjunction of symbolic constraints along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate further test inputs which would direct the program along alternative paths. Specifically, they systematically negate the conjuncts in the path constraint to provide a depth first exploration of all paths in the computation tree. If it is not feasible to solve the modified constraints, Godefroid et al. propose simply substituting random concrete values.

A challenge in applying Godefroid et al.'s approach is to provide methods which extract and solve the constraints generated by a program. This problem is particularly com-

CUTE: A Concolic Unit Testing Engine for C (ACM SIGSOFT Impact Award 2019)

Koushik Sen, UC Berkeley

Darko Marinov, UIUC

Gul Agha, UIUC

Thank you!